Impact of MVC and MVVM Architectures on Test Quality: An Analysis of GitHub Open-Source Repositories

Frederico Andrade
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Minas Gerais, Brazil
fredericoandrade7@gmail.com

Hugo de Paula
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Minas Gerais, Brazil
hugo@pucminas.br

Aline Brito
Department of Computing, Federal University of Ouro Preto (UFOP)
Ouro Preto, Minas Gerais, Brazil
aline.brito@ufop.edu.br

Leonardo Cardoso
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Minas Gerais, Brazil
leonardocardoso@pucminas.br

Cleiton Tavares
Department of Software Engineering, Pontifical Catholic University of Minas Gerais (PUC Minas)
Belo Horizonte, Minas Gerais, Brazil
cleitontavares@pucminas.br

ABSTRACT

This study compared the quality of test code in projects based on the *Model-View-Controller* (MVC) and *Model-View-ViewModel* (MVVM) architectures in GitHub repositories. The effectiveness of test suites was evaluated through mutation testing.The research problem lies in the lack of empirical evidence on whether architectural differences, which affect modularity, separation of concerns, and ease of test creation, also translate into higher effectiveness of test suites. To address this, the objective was to evaluate and compare the fault detection capability of tests in both architectures using mutation testing. The methodology involved computing mutation scores, examining the relationship between code coverage and test effectiveness, and analyzing the incidence of mutants that fail due to compile-time errors. The experiments were conducted with *Stryker.NET* for the automated execution of mutation tests and subsequently applied statistical analyses to interpret the collected data. The study showed no evidence of the impact of architectural choices on test effectiveness. Results also showed a mild correlation between code coverage of tests and mutation score. This finding is relevant, as it indicates that although more modular architectures facilitate the creation of tests, this characteristic does not necessarily translate into higher quality of the test suites.

KEYWORDS

MVC Architecture, MVVM Architecture, Test Quality

1 Introduction

The quality of testing is essential to ensure software reliability and maintainability by validating functionalities and detecting failures before they reach the end user [6]. Software architectures such as *Model-View-Controller* (MVC) and *Model-View-ViewModel* (MVVM) directly influence how tests are structured, impacting aspects such as modularity, coupling, and separation of concerns [8, 17]. Architectures that promote high modularity and low cohesion, as discussed by Gomaa [4], facilitate maintainability and contribute to more effective testing, reducing the long-term effort required to ensure test quality.

Although studies exist that explore the testability of architectures such as MVC and MVVM, most of these analyzes are limited to controlled scenarios and simplified prototypes, without considering the dynamics of real-world and collaborative projects such as open-source repositories [8, 17], where the management and maintenance of test code—carried out by multiple contributors are even more complex and essential for the continuous quality of the project [2]. This gap hinders a clear understanding of how these architectures influence test code quality in practical contexts.

Several studies have previously explored architectures such as *Model-View-Controller* (MVC), *Model-View-ViewModel* (MVVM), and *Model-View-Intent* (MVI), comparing them in terms of testability, maintainability, and performance [8]. However, most of these investigations are conducted using baseline projects specifically designed for measurement purposes, which limits the findings to controlled environments. Comparisons between MVVM and MVC in terms of testability and maintainability have been performed on graphical interface applications [17]; however, they do not extend to real and collaborative projects such as those found on GitHub.

In this context, the goal of this paper is to compare the *Model-View-Controller* (MVC) and *Model-View-ViewModel* (MVVM) architectures in terms of test quality in open-source projects. The specific goals are: (i) to evaluate and compare the fault detection rate (*mutation score*) between MVC and MVVM projects using mutation testing to measure the effectiveness of each architecture's test suite; (ii) to analyze test coverage effectiveness by combining the line coverage rate with the mutation score to assess whether test suite reach in MVC and MVVM projects is associated with higher fault detection capability; and (iii) to examine the proportion of mutants that result in compilation errors in each architecture, aiming to identify structural limitations or coupling issues that may compromise system testability.

At the end of the study, the results revealed that although MVVM projects exhibited higher median values in most metrics, including a median *Mutation Score* of 50.62% compared to 31.23% in MVC, these differences were not statistically significant. The high internal variability and heterogeneity across projects likely reduced the power of statistical tests to detect consistent effects. Moreover, no strong or significant correlation was found between repository age or line coverage and test effectiveness, as measured by mutation testing. This suggests that factors such as architecture, coverage, and maturity alone do not determine test quality in practice. These findings suggest that external factors, such as developer experience, testing culture, and project complexity, may have a more significant impact on the quality of test code in open-source contexts than the architecture itself. Thus, while architecture can support good testing practices, its impact on fault detection is not guaranteed in isolation, reinforcing the importance of adopting well-defined testing strategies regardless of the architectural model employed.

The remainder of this paper is organized into seven sections. Section 2 presents the theoretical foundation, covering the MVC and MVVM architectures and test quality concepts. Section 3 reviews related work, highlighting comparative studies and test quality metrics across different architectures. Section 4 describes the methodology used for collecting and analyzing data from open-source repositories. Section 5 introduces the artifacts developed for the study. Section 6 presents the results for each metric. Section 7 discusses the findings, and Section 8 presents the threats to validity. Finally, Section 9 concludes the study.

2 Background

This section presents the concepts of *Model-View-Controller* (MVC), *Model-View-ViewModel* (MVVM), Software Testing, and Mutation Testing. These concepts provide the theoretical foundation for analyzing the quality of test suites in repositories that adopt these architectures.

2.1 Model-View-Controller (MVC)

MVC is an architectural pattern designed for applications that interact with users through graphical interfaces, with its main advantage being the separation of concerns between the visual interface and the business logic of the application [7]. The pattern divides the system into three primary layers: *Model*, *View*, and *Controller*, as illustrated in Figure 1.

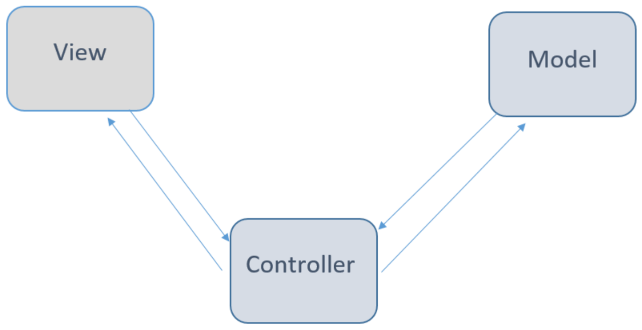

Figure 1: Representation of the MVC architecture [12].

The *Model* is the layer responsible for managing application data, interacting with the database, and applying business logic to that data. It encapsulates the logic for accessing and manipulating data without directly exposing these operations to other parts of the system. The *View* displays information to the user and represents the data and graphical interface layer. However, it does not directly communicate with the *Model*; all data flow is handled by the *Controller*. The *Controller* acts as an intermediary between the *View* and the *Model*, receiving user input, processing it, and determining how the data from the *Model* will be displayed in the *View* [7].

2.2 Model-View-ViewModel (MVVM)

The *Model-View-ViewModel* (MVVM) is an architectural pattern that evolved from MVC, aiming to simplify the development of user interfaces in complex applications [13]. This pattern splits the application into three main components: *Model*, *ViewModel*, and *View*, as shown in Figure 2.

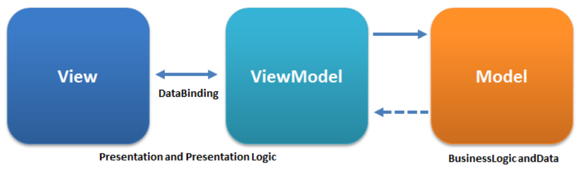

Figure 2: Representation of the MVVM architecture [16].

MVVM shares similarities with MVC but introduces the *View-Model* layer, which is responsible for managing the state of the interface (*View*) and user interactions without requiring the *View* to be aware of the business logic (*Model*). This separation is made possible by *Data Binding*, which enables automatic synchronization between the interface data and the application state, such that any change in one is immediately reflected in the other. As a result, the UI developer does not need to understand the internal workings of the *View*, ensuring greater independence between classes and consequently enhancing interoperability and the creation of unit tests [13, 17].

2.3 Software Testing and Mutation Testing

In scenarios where software errors can lead to critical failures, such as disruptions in financial services or security issues, testing plays a key role in validating functionality and preventing faults [6]. However, traditional testing approaches, such as unit and integration tests, often fail to detect more complex issues related to business logic or edge conditions. To address these limitations, mutation testing offers an alternative that assesses test suite robustness by introducing small, controlled modifications to the code to simulate potential defects [15].

These controlled modifications, called “mutants,” simulate potential faults by altering conditional operators, return values, or variable assignments. The effectiveness of a test suite is measured by its ability to detect and "kill" these mutants, meaning it can identify the injected modifications. One approach to assessing the quality of mutation testing is to use a mutation score. The *Mutation Score* metric expresses the percentage of killed mutants relative to the total generated and serves as a direct indicator of test robustness [14, 15]. In architectures such as MVC and MVVM, which promote the separation of concerns, mutation testing is particularly useful for exploring structural differences that affect testability, such as layer isolation and component modularity [17].

3 Related Work

The related work explored in this section addresses the context of software quality in testing, as well as comparative studies between architectures such as MVVM and MVC based on performance-oriented quality.

Wu et al. [20] investigated the impact of eliminating *test smells* on code quality by analyzing 119 versions of 10 open-source projects. The authors identified and removed five types of test smells: *Mystery Guest*, *Resource Optimism*, *Eager Test*, *Assertion Roulette*, and *Sensitive Equality*, quantifying the results with metrics such as *defect-proneness* and *change-proneness*. Their findings showed that refactoring test smells significantly reduces the likelihood of defects and changes in both test and production code. This study is relevant to the present work as it demonstrates how specific characteristics of test code can influence its effectiveness, particularly in projects that adopt different architectural structures such as MVC and MVVM.

Wisnuadhi et al. [18] compared the performance of MVP and MVVM architectures in Android applications, evaluating metrics such as CPU usage, memory consumption, and execution time. A point-of-sale application was implemented for this comparison, revealing that MVVM outperformed MVP in CPU efficiency and execution time, while MVP showed better performance in memory usage. Although the focus was on performance, the analysis method offers insight into how different architectures can be practically compared. This work contributes methodologically by highlighting the importance of evaluating specific architectural aspects, which can be adapted to study differences in test quality between MVC and MVVM.

Jun and Rana [5] assessed the impact of design patterns such as *Factory Method*, *Singleton*, and *Decorator* on code maintainability, using tools like *SonarQube*. Their results indicated that well-implemented design patterns increase modularity and facilitate maintenance, helping to reduce code complexity. The relevance of this study lies in demonstrating how the adoption of structural approaches affects overall code quality, allowing one to infer how MVC and MVVM architectures may influence test organization and effectiveness.

Sholichin et al. [11] conducted a comparative review of iOS architectural patterns, including MVC, MVP, MVVM, and VIPER, using metrics such as cohesion, coupling, and resource consumption. Their analysis showed that MVVM stood out in testability and cohesion, while VIPER demonstrated greater efficiency in coupling and CPU usage. This study is particularly relevant for applying practical metrics to evaluate how architectures influence software quality, complementing the objectives of the present study by exploring the impact of architectures on test quality.

Sánchez et al. [14] investigated developers' perceptions of mutation testing in open-source projects. The study revealed that the technique is widely valued for improving fault detection and increasing test effectiveness; however, its use in larger projects faces challenges related to tool performance. This work reinforces the relevance of mutation testing as an analytical methodology, directly aligning with the goals of the present study by using the technique to explore architectural differences and understand testing effectiveness in real-world contexts.

4 Study Design

This study adopted a quantitative approach, mining and analyzing GitHub repositories to compare the quality of unit tests in projects using the MVC and MVVM architectures. The following sections outline the study's steps, including data collection methods, analysis procedures, tools, and processes employed to achieve the objectives. Figure 3 shows the steps used to conduct the study.

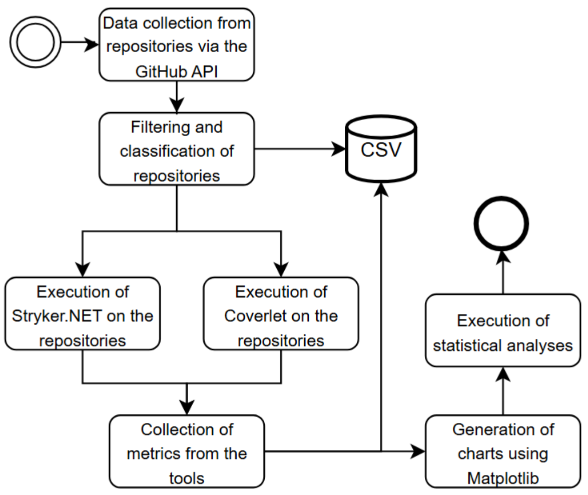
Data collection from repositories via the GitHub API
Filtering and classification of repositories
CSV
Execution of Stryker.NET on the repositories
Execution of Coverlet on the repositories
Collection of metrics from the tools
Generation of charts using Matplotlib
Execution of statistical analyses

Figure 3: Study Steps

4.1 Repository Selection and Classification

Repositories were collected via GitHub's *GraphQL API*, using a Python script to select public C# projects that specify *TargetFramework* as *net6.0*, *net8.0*, or *net9.0*. Versions *.NET 8* and *.NET 9* were chosen for their ongoing support, while *.NET 6* was selected for being a recent LTS version with extended support [9]. The repositories have more than 100 stars and were collected monthly between 2010 and 2024 to ensure a representative sample [3]. To confirm the presence of tests, *.csproj* files were analyzed for packages such as *NUnit*, *xUnit*, *Test*, and *MSTest*.

Repositories were classified as MVC or MVVM based on dependency analysis in their *.csproj* files, using specific criteria to identify packages associated with each architecture. Repositories were labeled as MVC when they contained references to *Microsoft.AspNetCore.Mvc* or *System.Web.Mvc*, which are widely recognized as implementations of the *Model-View-Controller* pattern in .NET applications. Conversely, repositories including dependencies such as *CommunityToolkit.Mvvm*, *MvvmLight*, or *ReactiveUI* were categorized as MVVM, due to their usage in implementing the *Model-View-ViewModel* pattern. These frameworks play a crucial role in structuring the project, influencing code organization, separation of concerns, and maintainability. To ensure classification reliability and precision, repositories lacking references to any of these packages were discarded. In hybrid cases, where dependencies from both patterns were present, classification was based on the first framework identified in the *.csproj* file, regardless of subsequent frameworks.

4.2 Tools

To analyze test quality and mutation coverage, the tool used was *Stryker.NET*[1]. It enabled the assessment of test robustness by introducing small code changes (mutants) and measuring the effectiveness of the tests in identifying these changes.

Additionally, to calculate test coverage in the repositories, the tool *Coverlet*[2] was employed. Coverlet measures code coverage rates, providing line, method, and branch coverage data, which complement mutation analysis by offering insights into how much code is being tested.

4.3 Metrics and Evaluation

The metrics selected to assess test quality in MVC and MVVM architectures were chosen to explore different aspects of test suite effectiveness with respect to failure detection. In addition to these core metrics, complementary data such as code coverage and repository age were considered to provide a broader overview of the analyzed projects. The main metrics are described below:

I. **Fault Detection Rate (*Mutation Score*):** This metric calculates the percentage of eliminated mutants in relation to the total number of introduced mutants during mutation testing. Higher values indicate greater effectiveness of the test suites in identifying simulated faults. Comparing this score between MVC and MVVM repositories allowed for the assessment of each architecture's fault detection capability.
II. **Test Coverage Effectiveness Rate:** This metric assesses test coverage effectiveness with respect to mutation detection. By combining code coverage rates with the mutation score, it offers a more accurate view of test quality, ensuring that comparisons between architectures consider both reach and fault detection capacity.
III. **Proportion of Mutants with Compilation Errors:** This metric indicates the relative number of mutants that could not be tested due to compilation errors after code modifications. A high number of such mutants may point to rigid code structures or excessive coupling, making the system more sensitive to internal changes. Although these mutants do not directly affect the mutation score, their occurrence may reflect structural challenges that negatively impact testability.

4.4 Data Analysis and Presentation

The analysis of the collected data was performed using graphical representations and statistical tests to identify potential differences between MVC and MVVM repositories. Furthermore, correlations between line coverage and mutation score, as well as between repository age and mutation score, were investigated.

To verify whether the data followed a normal distribution, the *Shapiro-Wilk* test was applied. Based on the outcome, the appropriate statistical test was selected to compare the metrics across architectures. If the data followed a normal distribution, a *Student's t-test* was used; otherwise, the non-parametric *Mann-Whitney* test was applied. The statistical analysis considered a confidence level of 95% ($\alpha = 0.05$).

Additionally, a linear regression was performed to quantify the relationship between line coverage and mutation score, as well as between repository age and mutation score. For this, the *linregress* function from the *SciPy* library[3] was used, which returns, among other parameters, the coefficient of determination (r) [10].

5 Development

This section describes the development of the metric collection process from the repositories, detailing the implemented procedures and the data gathered for analysis.

5.1 Mutation Testing Execution

To perform mutation testing, a script was developed to clone each repository, restore its dependencies, and compile the code before running *Stryker.NET*. Each repository was validated for the presence of a *.sln* solution file. If any step failed, the errors were logged and the repository was excluded from the analysis. After environment setup, the script executed the commands *dotnet restore* and *dotnet build* to ensure that dependencies were properly configured and the code compiled without errors. *Stryker.NET* was then triggered to generate metrics such as *Killed*, *Survived*, *Timeout*, *Time Elapsed*, and *Mutation Score*, which were stored in a CSV file for later analysis.

5.2 Code Coverage Analysis

In addition to mutation testing, another script was implemented to evaluate the unit test coverage in the classified repositories. Using the *Coverlet* tool, the script automated the execution of the *dotnet test* command to identify the generated *.dll* files, which are essential for measuring code coverage. The output directories from the build process (*bin/Debug* and *bin/Release*) were analyzed to locate these files. If no detectable tests were found in a repository, it was removed from the analysis. After identifying the test files, *Coverlet* was triggered to calculate the coverage of lines and methods tested. The results were extracted from the *Coverlet* output and stored in a CSV file containing metrics such as "Line Coverage (%)" and "Method Coverage (%)".

6 Results

This section presents the results obtained from the analysis of the selected repositories: 53 MVC projects and 17 MVVM projects. To verify the normality of the metric distributions, the *Shapiro-Wilk* test was applied. The results showed that, except for the *Mutation Score* in MVVM repositories ($p = 0.0511$), the other distributions do not follow a normal distribution ($p \leq 0.05$). Therefore, the *Mann-Whitney U* test was chosen for group comparison, as it is a suitable non-parametric test for this scenario.

[1]Stryker.NET is an open-source tool designed to perform mutation testing on .NET-based projects. More information at: https://stryker-mutator.io
[2]Coverlet is a cross-platform code coverage framework for *.NET* that supports line, branch, and method coverage. More information at: https://github.com/coverlet-coverage/coverlet

[3]https://scipy.org/

Figure 4 shows the distribution of the *Mutation Score* between MVC and MVVM repositories. MVVM projects had a median of 50.62%, considerably higher than the median of MVC projects, which was 31.23%, possibly indicating greater effectiveness of the test suites in the MVVM architecture. Furthermore, MVVM values exhibited greater dispersion, with an interquartile range of 61.74 ($Q1 = 9.56; Q3 = 71.30$), compared to the 45.16 range observed in MVC projects ($Q1 = 10.25; Q3 = 55.40$). Both groups, however, presented high variability. Despite the visual and interquartile differences, the *Mann-Whitney U* test indicated that this difference is not statistically significant ($p = 0.6609$).

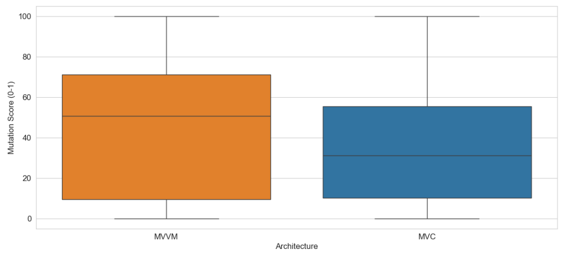

Figure 4: *Mutation Score* by Architecture

Figure 5 shows the distribution of line coverage rate between MVC and MVVM repositories. Both groups showed high and similar dispersion, with values ranging from approximately 0 to nearly 1. The median coverage in MVC projects was slightly higher (0.344) than that in MVVM projects (0.321), suggesting somewhat broader coverage in the first group. Regarding dispersion, MVC repositories had an interquartile range of 0.653 ($Q1 = 0.092; Q3 = 0.745$), while MVVM projects had a range of 0.603 ($Q1 = 0.128; Q3 = 0.731$), indicating similar amplitudes between groups. Despite these visual differences, the *Mann-Whitney U* test did not indicate any statistically significant difference between the two groups ($p = 0.8052$).

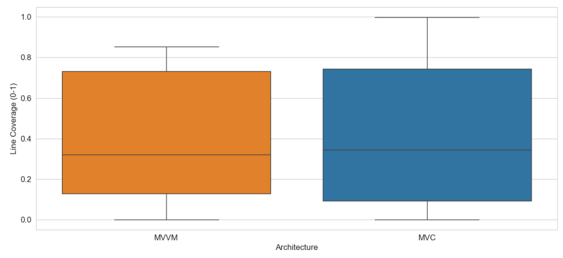

Figure 5: Line Coverage by Architecture

Figure 6 presents the relationship between test coverage and *Mutation Score* in the analyzed repositories, with linear regression lines separated by architecture. Projects with low coverage generally had lower *Mutation Scores*. In repositories with high coverage, the *Mutation Score* varied widely. Linear regression indicated a moderate positive correlation between the variables ($r^2 = 0.52$) in

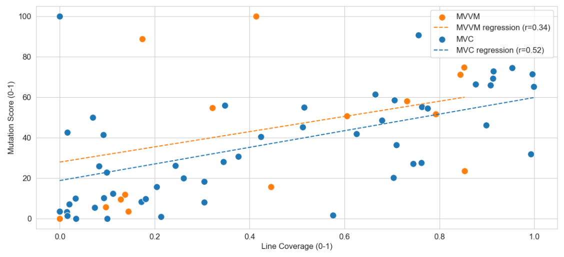

Figure 6: Relationship between Line Coverage and Mutation Score

MVC projects, while in MVVM projects the observed correlation was moderate-weak ($r^2 = 0.34$).

Figure 7 shows the distribution of the proportion of mutants with compilation errors between MVC and MVVM repositories. In both groups, most values are concentrated below 10%, although some outliers are present. The median of MVVM projects was slightly higher (5.13%) compared to MVC projects (4.58%). Regarding dispersion, MVC repositories showed an interquartile range of 4.11 ($Q1 = 2.02; Q3 = 6.13$), while MVVM projects had a range of 3.53 ($Q1 = 4.39; Q3 = 7.92$), indicating similar variations between the groups. Despite these specific differences, the *Mann-Whitney U* test did not identify any statistically significant difference between the architectures ($p = 0.1977$).

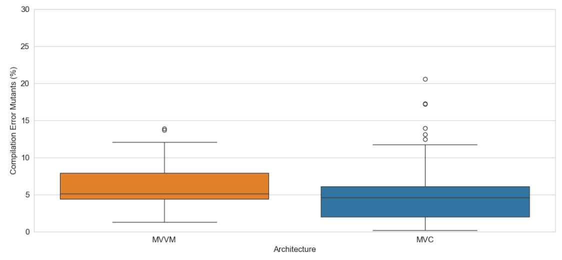

Figure 7: Mutants with Compilation Errors

Figure 8 shows the relationship between repository age (in years) and *Mutation Score*. A greater concentration of projects with low *Mutation Scores* is observed in repositories up to five years old. In older repositories, the data is more dispersed. The distribution of the points does not show a clear pattern between the variables, with outliers present across different age ranges. The two architectures are distributed throughout the chart, overlapping in several regions. Correlation analyzes revealed a determination coefficient of $r^2 = 0.17$ for both MVC and MVVM projects, indicating a weak correlation between repository age and *Mutation Score* in both cases.

7 Discussions

The results presented in Section 6 indicated that there were no statistically significant differences between the MVC and MVVM

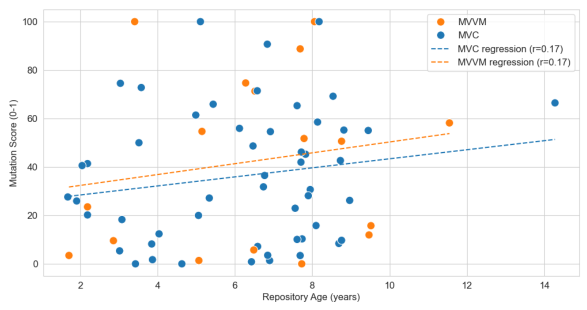

Figure 8: Relationship between Repository Age and Mutation Score

architectures with respect to the analyzed metrics: fault detection rate (*Mutation Score*), line coverage, and the proportion of mutants with compilation errors. The analyses performed using the *Mann-Whitney U* test pointed to a lack of sufficient evidence to affirm that architecture, in isolation, influences the effectiveness of test suites in these projects.

Nevertheless, descriptive analyzes revealed that MVVM projects had higher medians across most metrics, especially the *Mutation Score*, which had a median of 50.62% compared to 31.23% in MVC projects. Additionally, the absence of a well-defined central tendency was observed, with wide interquartile ranges: 62.06 percentage points for MVVM and 47.78 for MVC in the *Mutation Score* by architecture chart, indicating high internal variability within both groups—especially considering that the metric ranges from 0% to 100%. Such variability may have directly contributed to the non-significant statistical results, as highly heterogeneous distributions reduce the power of statistical tests to detect actual differences.

These findings contrast with the theoretical expectation that more modular architectures, such as MVVM, would favor testability by facilitating component isolation and the creation of more targeted tests [17]. The empirical data collected in this study suggest that such an architectural advantage, if it exists, did not consistently manifest in the objective metrics adopted herein.

Regarding repository age, the results also did not indicate a strong relationship with the *Mutation Score*. The linear regression determination coefficient (r^2) was 0.17 for both architectures, representing a weak correlation between the variables. Although older repositories presented greater dispersion in *Mutation Score* values, this variation was not sufficient to establish a clear trend. This suggests that project temporal maturity, in isolation, is also not a determining factor for test quality as measured by the adopted metrics.

In terms of line coverage and *Mutation Score*, the results also did not reveal a statistically significant association between the variables. The linear regression for MVC projects yielded a determination coefficient of 0.52, a value similar to that found by Assylbekov et al. [1], who identified a moderate correlation ($r^2 = 0.5$) between these metrics in a study with open-source projects. For MVVM projects, the correlation was weaker ($r^2 = 0.34$). Despite the similarity between the studies, the lack of statistical significance in the current work suggests that this observed association was not robust enough to indicate a strong relationship. Just as with repository age, line coverage—when considered in isolation—was also not a determining factor for test quality under the analyzed conditions.

One possible explanation, as also discussed by Assylbekov et al. [1], lies in the influence of uncontrolled factors such as developer experience, project maturity, and the testing culture present in each team. In collaborative environments like GitHub, these elements tend to vary significantly between projects, diluting potential systematic effects attributed to architecture. Moreover, the analyzed repositories span diverse domains, with sizes varying by orders of magnitude, and present significant differences in complexity and adopted tools, which may have impacted the results in different ways.

It is worth noting that although the adopted metrics provide a quantitative assessment of test effectiveness, they do not capture other important aspects such as readability, granularity, or maintainability of test suites. Therefore, the results presented here do not rule out the possibility that architecture influences test quality, given that the median *Mutation Score* for MVVM was approximately 19% higher than that of MVC. However, the results indicate that, in the observed context, such influence was not statistically evident due to the high variability and diversity of the analyzed projects.

8 Threats to Validity

This section presents the main threats to the validity of this study, along with the strategies adopted to minimize them[19].

Internal Validity. Internal validity refers to the extent to which the results of the experiment can be attributed to the studied variables, without the influence of uncontrolled external factors. A relevant threat lies in the differences in size, scope, and complexity among the analyzed projects. Larger projects tend to accumulate more legacy code, present greater diversity in testing practices, and be subject to failures unrelated to architecture. Although no explicit normalization of these factors was performed, the selection included only repositories with active automated tests that compile and execute correctly, filtered by language, technology, and popularity, partially reducing heterogeneity across projects.

Conclusion Validity. Conclusion validity concerns the robustness of statistical inferences drawn from the collected data. One limitation is the use of *Mutation Score* as the primary metric for assessing test quality, which does not include other potential indicators such as error severity, number of test cases, test coverage, and root cause. To mitigate this limitation, complementary metrics such as code coverage were also analyzed. Another important limitation is the reduced number of MVVM repositories available, which likely decreased the statistical power of the analyses. This suggests that the absence of statistically significant differences between MVC and MVVM may have been influenced by the study design, particularly the sample size, which could have prevented their detection.

External Validity. External validity relates to the generalizability of the study results to other contexts. As the analysis was restricted to open-source projects hosted on GitHub, there is a risk that the results do not adequately represent corporate environments or closed platforms. Public projects may adopt development and testing practices that differ from those in private organizations, both in terms of culture and operational constraints. To mitigate this limitation, the selection included popular repositories spanning various domains and development realities.

Construct Validity. Construct validity refers to how accurately the investigated concepts were operationalized and measured. A limitation lies in the technical constraints of the Stryker.NET tool, which may not support all coding patterns present in the analyzed repositories, potentially generating invalid or non-representative mutants. This occurs because certain language constructs or dependencies from specific frameworks may not be correctly instrumented by the tool. To reduce the impact of this limitation, a manual review of Stryker.NET-generated reports was performed to identify and discard invalid or non-representative mutants. Another limitation lies in the strategy of classifying repositories as MVC or MVVM based on the first architectural dependency found in `.csproj` files, even when other patterns were also present. This approach may introduce bias in projects using multiple architectural styles, but it was adopted as a way to reduce subjectivity in classification and enable large-scale automation of the analysis.

9 Conclusion

This study conducted a comparative analysis of test code quality in projects that adopt the MVC and MVVM architectures, using public repositories from GitHub. It is worth noting that this study focused exclusively on projects developed in the C# language and the *.NET framework*. The main evaluation metric was the *Mutation Score*, obtained through the execution of the *Stryker.NET* tool. In addition to this metric, complementary information such as code coverage was considered, allowing for a broader evaluation of the effectiveness of the tests present in the analyzed projects.

When evaluating and comparing the fault detection rate between the MVC and MVVM architectures, it was observed that, although the median *Mutation Score* of MVVM projects was higher than that of MVC projects, this difference did not reach statistical significance. Therefore, it was not possible to confirm that architecture directly impacts the effectiveness of tests in terms of fault detection. The analysis of the relationship between line coverage and the *Mutation Score* revealed a moderate positive correlation for MVC projects and a weak one for MVVM, but similarly with no statistical significance, indicating that coverage alone does not guarantee greater fault detection capability. Finally, the investigation of the proportion of mutants with compilation errors showed similar values between architectures, with no statistically relevant differences, suggesting no evidence of structural limitations affecting system testability.

These results indicate that, despite the theoretical advantages often attributed to MVC and MVVM architectures in terms of modularity and testability, the conducted analysis did not reveal a direct influence of architectural choice on test quality in open-source projects. External factors, such as developer experience, project maturity, and testing culture, appear to have a potentially more significant impact. As a practical conclusion, the data suggest that the quality of test suites is less related to the adopted architecture and more to how tests are planned, written, and maintained by the teams involved. Therefore, architectural decisions should be accompanied by a conscious testing strategy, as architecture alone does not ensure greater fault detection effectiveness.

As a proposal for future work, a promising direction would be to investigate the influence of developer experience and practices on the quality of test code, through the analysis of contributor profiles in the repositories, considering their seniority, frequency of contribution, and involvement with automated testing. Such an approach would allow for a better understanding of whether test quality is more closely linked to human and organizational factors than strictly to the employed architecture.

ARTIFACT AVAILABILITY

The *scripts* and *dataset* are available at: *https://github.com/CleitonSilvaT/mvc-mvvm-sbqs-2025*

REFERENCES

[1] Berik Assylbekov, Erick Gaspar, Nasir Uddin, and Paul Egan. 2013. Investigating the Correlation between Mutation Score and Coverage Score. In *2013 UKSim 15th International Conference on Computer Modelling and Simulation*. 347–352. doi:10.1109/UKSim.2013.28

[2] Matthijs Besten, Jean-Michel Dalle, and Fabrice Galia. 2006. Collaborative Maintenance in Large Open-Source Projects, In IFIP International Federation for Information Processing. *International Federation for Information Processing Digital Library; Open Source Systems* 203. doi:10.1007/0-387-34226-5_23

[3] Hudson Borges and Marco Tulio Valente. 2018. What's in a GitHub Star? Understanding Repository Starring Practices in a Social Coding Platform. *Journal of Systems and Software* 146 (2018), 112–129. doi:10.1016/j.jss.2018.09.016

[4] Hassan Gomaa. 2011. *Software Quality Attributes*. Cambridge University Press, 357–368.

[5] Hen Kian Jun and Muhammad Ehsan Rana. 2021. Evaluating the Impact of Design Patterns on Software Maintainability: An Empirical Evaluation. In *2021 Third International Sustainability and Resilience Conference: Climate Change*. 539–548. doi:10.1109/IEEECONF53624.2021.9668025

[6] Ali Khatami and Andy Zaidman. 2023. Quality Assurance Awareness in Open Source Software Projects on GitHub. In *2023 IEEE 23rd International Working Conference on Source Code Analysis and Manipulation (SCAM)*. 174–185. doi:10.1109/SCAM59687.2023.00027

[7] A. Leff and J.T. Rayfield. 2001. Web-application development using the Model/View/Controller design pattern. In *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*. 118–127. doi:10.1109/EDOC.2001.950428

[8] Hannelore Magics-Verkman, Doina Rodica Zmaranda, Cornelia Aurora Győrödi, and Robert-Stefan Győrödi. 2023. A Comparison of Architectural Patterns for Testability and Performance Quality for iOS Mobile Applications Development. In *2023 17th International Conference on Engineering of Modern Electric Systems (EMES)*. 1–4. doi:10.1109/EMES58375.2023.10171619

[9] Microsoft. 2025. .NET and .NET Core Support Policy. https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-core. [Online; accessed 29-March-2025].

[10] SciPy Documentation. 2023. scipy.stats.linregress — SciPy v1.11.3 Manual. https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.linregress.html

[11] Fauzi Sholichin, Mohd Isa, Shahliza Abd Halim, and M Firdaus Harun. 2019. Review of iOS Architectural Pattern for Testability, Modifiability, and Performance Quality. *Journal of Theoretical and Applied Information Technology* 97 (08 2019).

[12] Zanfina Svirca. 2020. Everything you need to know about MVC architecture. https://towardsdatascience.com/everything-you-need-to-know-about-mvc-architecture-3c827930b4c1. [Online; accessed 12-October-2024].

[13] Artem Syromiatnikov and Danny Weyns. 2014. A Journey through the Land of Model-View-Design Patterns. In *2014 IEEE/IFIP Conference on Software Architecture*. 21–30. doi:10.1109/WICSA.2014.13

[14] Ana B. Sánchez, José A. Parejo, Sergio Segura, Amador Durán, and Mike Papadakis. 2024. Mutation Testing in Practice: Insights From Open-Source Software Developers. *IEEE Transactions on Software Engineering* 50, 5 (2024), 1130–1143. doi:10.1109/TSE.2024.3377378

[15] Victor Veloso and Andre Hora. 2022. Characterizing high-quality test methods: a first empirical study. In *Proceedings of the 19th International Conference on Mining Software Repositories* (Pittsburgh, Pennsylvania) *(MSR '22)*. Association for Computing Machinery, New York, NY, USA, 265–269. doi:10.1145/3524842.3529092

[16] Wikipedia. 2024. Model–view–viewmodel — Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.phptitle=Model%E2%80%93view%E2%80%

93viewmodel&oldid=1239976501. [Online; accessed 13-October-2024].

[17] Amy Wilson, Fadi Wedyan, and Safwan Omari. 2022. An Empirical Evaluation and Comparison of the Impact of MVVM and MVC GUI Driven Application Architectures on Maintainability and Testability. In *2022 International Conference on Intelligent Data Science Technologies and Applications (IDSTA)*. 101–108. doi:10.1109/IDSTA55301.2022.9923083

[18] Bambang Wisnuadhi, Ghifari Munawar, and Ujang Wahyu. 2020. Performance Comparison of Native Android Application on MVP and MVVM. In *International Seminar of Science and Applied Technology (ISSAT 2020)*. doi:10.2991/aer.k.201221.047

[19] Claes Wohlin, Per Runeson, Martin Hst, Magnus C. Ohlsson, Bjrn Regnell, and Anders Wessln. 2012. *Experimentation in Software Engineering*. Springer Publishing Company, Incorporated.

[20] Haitao Wu, Ruidi Yin, Jianhua Gao, Zijie Huang, and Huajun Huang. 2022. To What Extent Can Code Quality be Improved by Eliminating Test Smells?. In *2022 International Conference on Code Quality (ICCQ)*. 19–26. doi:10.1109/ICCQ53703.2022.9763153